YOBE STATE UNIVERSITY, DAMATURU

Km 7 Gujba Road P.M.B. 1144, Damaturu. Yobe State. Nigeria

DEPARTMENT OF COMPUTER SCIENCE

CSC4421
ALGORITHMS AND COMPLEXITY ANALYSIS

2024/2025 Academic Session

Course Content

Introduction

Basic algorithmic analysis: Asymptotic analysis of Upper and average complexity bounds; standard Complexity Classes Time and space tradeoffs in algorithms analysis, recursive algorithms.

Algorithmic Strategies: Fundamental computing algorithms: Numerical algorithms, sequential and binary search algorithms; sorting algorithms, Binary Search tress, Hash tables, graphs & its representation.

References:

1. Thomson H, Cormen, Leiserson, Rivest and Stein, “Introductions to Algorithms”, PHI
2. Alfred V. Aho, Jeffrey D, Ullman and John E. Hopcroft, Data Structures and Algorithms

Introduction

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms;

- Search − Algorithm to search an item in a data structure.
- Sort − Algorithm to sort items in a certain order.
- Insert − Algorithm to insert item in a data structure.
- Update − Algorithm to update an existing item in a data structure.
- Delete − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics;

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code. As we know that all programming languages share basic **code constructs** like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution. Let's take an example of the **Problem** − Design an algorithm to add two numbers and display the result.

```
Step 1 – START
Step 2 – declare three integers a, b & c
Step 3 – define values of a & b
Step 4 – add values of a & b
Step 5 – store output of step 4 to c
Step 6 – print c
Step 7 – STOP
```

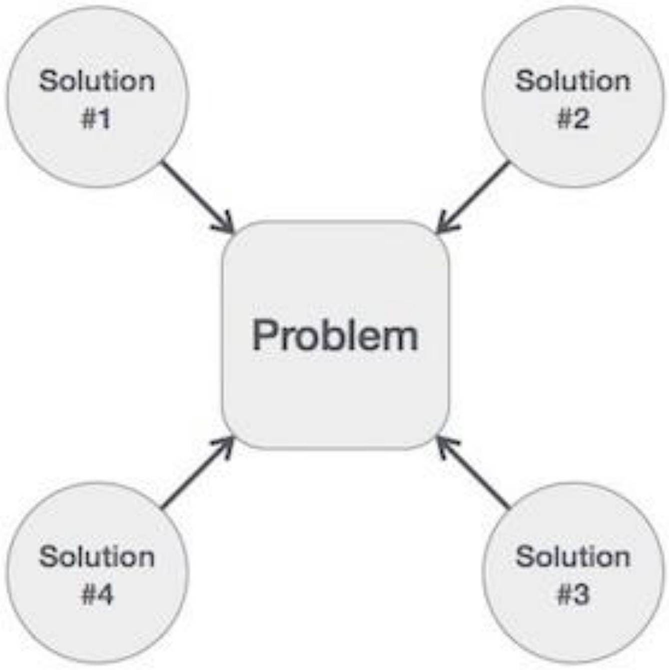
Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as;

```
Step 1 – START ADD
Step 2 – get values of a & b
Step 3 – c ← a + b
Step 4 – display c
Step 5 – STOP
```

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one way.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Analysis of the algorithm generally focused on CPU (time) usage, Memory usage, Disk usage, and Network usage. All are important, but the most concern is about the CPU time.

Be careful to differentiate between:

Performance: How much time/memory/disk/etc. is used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.

Complexity: How do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger.

Note: Complexity affects performance but not vice-versa.

Algorithm Analysis:

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why Analysis of Algorithms is important?

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following;

A Priori Analysis − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.

A Posterior Analysis − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Types of Algorithm Analysis:

Best case: Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.

Worst Case: Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is not present at all then the worst case occurs.

Average case: In the average case take all random inputs and calculate the computation time for all inputs, and then we divide it by the total number of inputs.

Algorithm Complexity

Algorithm means any technique that can be used to solve a given problem. The problem under concern could be that of rearranging a given sequence of numbers, solving a system of linear equations, finding the shortest path between two nodes in a graph, etc. An algorithm consists of a sequence of basic operations such as addition, multiplication, comparison, and so on and is typically described in a machine independent manner.

When an algorithm gets coded in a specified programming language such as C, C++, or Java, it becomes a program that can be executed on a computer. For any given problem, there could possibly be many different techniques that solve it. Thus, it becomes necessary to define performance measures that can be used to judge different algorithms. Two popular measures are the **time complexity** and the **space complexity.**

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

Time complexity

The time complexity or the run time of an algorithm refers to the total number of basic operations performed in the algorithm. Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases. Another example is to consider the problem of finding the minimum of n given numbers. This can be accomplished using n – 1 comparison.

Of the two measures perhaps, time complexity is more important. This measure is useful for the following reasons.

1) We can use the time complexity of an algorithm to predict its actual run time when it is coded in a programming language and run on a specific machine.
2) Given several different algorithms for solving the same problem we can use their run times to identify the best one.

Space Complexity

The space complexity of an algorithm is defined to be the amount of space (i.e., the number of memory cells) used by the algorithm. Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. This measure can be critical especially when the input data itself is huge.

The space required by an algorithm is equal to the sum of the following two components;

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept;

```
Algorithm: SUM(A, B)

Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop
```

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

We define the input size of a problem instance to be the amount of space needed to specify the instance. For the problem of finding the minimum of n numbers, the input size is n, since we need n memory cells, one for each number, to specify the problem instance. For the problem of multiplying two n × n matrices, the input size is $2n^2$, because there are these many elements in the input.

Both the run time and the space complexity of an algorithm are expressed as functions of the input size. For any given problem instance, its input size alone may not be enough to decide its time complexity.

To illustrate this point, consider the problem of checking if an element x is in an array `a[1:n]`. This problem is called the searching problem. One way of solving this problem is to check;

`if x = a[1];` if not check `if x = a[2]; and so on.`

This algorithm may terminate after the first comparison, after the second comparison, . . ., or after comparing x with every element in `a[]`. Thus, it is necessary to qualify the time complexity as the best case, the worst case, the average case, etc.

The average case run time of an algorithm is the average run time taken over all possible inputs

(of a given size). Analysis of an algorithm can be simplified using asymptotic functions such as

`O(.), Ω(.),` etc.

Let f(n) and g(n) be nonnegative integer functions of n.

We say

$f(n)$ is $O(g(n))$ if $f(n) \le c\, g(n)$ for all $n \ge n_0$ where c and n_0 are some constants.

Also, we say

$f(n) = \Omega(g(n))$ if $f(n) \ge c\, g(n)$ for all $n \ge n_0$ for some constants c and n_0.

If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$,

then we say

$f(n) = \theta(g(n))$.

Usually we express the run times (or the space complexities) of algorithms using $\theta()$. The algorithm for finding the minimum of n given numbers takes $\theta(n)$ time.

An algorithm designer is faced with the task of developing the best possible algorithm (typically an algorithm whose run time is the best possible) for any given problem.

Unfortunately, there is no standard recipe for doing this. Algorithm researchers have identified a number of useful techniques such as the divide-and-conquer, dynamic programming, greedy, backtracking, and branch-and-bound.

Application of any one or a combination of these techniques by itself may not guarantee the best

possible run time. Some innovations (small and large) may have to be discovered and incorporated.

Basic algorithmic analysis

The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared. An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change. The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis. The complexity or difficulty of any task is determined by how much time it takes for its completion. While writing an algorithm, we need to make sure it works fastest with the given resources. We can calculate the best, average, and worst-case scenarios for an algorithm by using asymptotic analysis.

Asymptotic Analysis of Algorithms

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations. Asymptotic notations are used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case. But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case. When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

In computer programming, the asymptotic analysis tells us the execution time of an algorithm. The lesser the execution time, the better the performance of an algorithm is. For example, let's assume we have to add an element at the starting of an array. Since an array is a contiguous memory allocation, we cannot add an element at the first position directly. We need to shift each element to the next position and then only we can add elements at the first position. So, we need to traverse the whole array once. The greater the array size, the longer the execution time. While performing a similar operation on a linked list is comparatively very easy since in a linked list each node is in connection with the next node via a pointer. We can just create a new node and point it to the first node of a linked list. So, you can see that adding an element at the first position in a linked list is very much easier as compared to performing a similar operation on an array. We similarly compare different data structures and select the best possible data structure for an operation. The less the execution time, the higher the performance.

Then, how do we find the time complexity of an algorithm. Computing the real running time of a process is not feasible. The time taken for the execution of any process is dependent on the size of the input. For example, traversing through an array of 5 elements will take less time as compared to traversing through an array of 500 elements. We can observe that time complexity depends on the input size.

Hence, **if the input size is 'n', then f(n) is a function of 'n'** denoting the time complexity. Calculating f(n) value for small programs is easy as compared to bigger programs. We usually compare data structures by comparing their f(n)values. We also need to find the growth rate of an algorithm because in some cases there is a possibility that for a smaller input size, one data structure is better than the other but when the input data size is large, it's vice versa.

For example: Let's assume a function $f(n) = 8n^2 +5n+12$.

Here n represents the number of instructions executed.

If n=1:

Percentage of time taken due to $8n^2 = (8/(8+5+12))*100 = 32\%$

Percentage of time taken due to $5n = (5/(8+5+12))*100 = 20\%$

Percentage of time taken due to $12 = (12/(8+5+12))*100 = 48\%$

In the above example, we can see that most of the time is taken by '12'. But based on only one example we cannot conclude time complexity. We have to calculate the growth factor and then observe the situation as we did in the table below.

N	**8N2**	**5N**	**12**
1	32%	20%	48%
10	92.8%	5.8%	1.4%
100	99.36%	0.62%	0.015%
1000	99.93%	0.06%	≈ 0 %

In the above table, we can observe that the $8n^2$ term is contributing most of the time. It is so much greater than the other two terms that we can ignore those two terms. Therefore, the complexity for this algorithm is:

$$F(n) = 8n^2$$

This is the approximate time complexity which is very close to the actual result. This measure of approximate time complexity is known as asymptotic complexity.

Here, we are considering the term which takes most of the time and eliminating the unnecessary terms. Though, we're not calculating the exact running time still it is very close to it.

The time required for the execution of an algorithm is categorized into three types:

- **Worst case:** the maximum time required for the execution
- **Average case:** average time taken for the execution.
- **Best case:** minimum time taken for the execution.

Asymptotic Notation

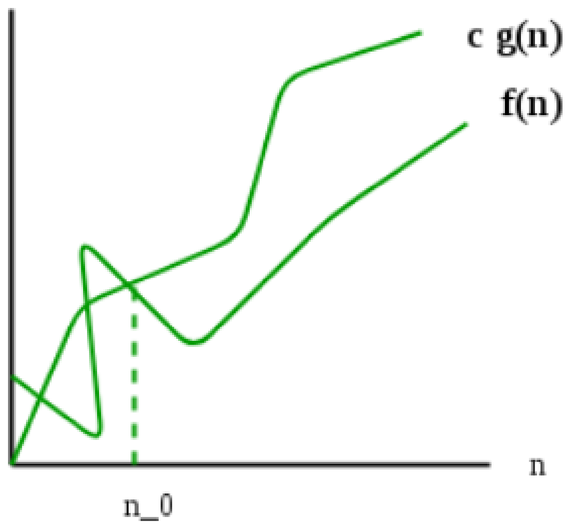
Commonly used asymptotic notation for representing the runtime complexity of an algorithm are:

- Big O notation (O)
- Omega notation (Ω)
- Theta notation (θ)

1. Big O notation (O)

This asymptotic notation measures the performance of an algorithm by providing the order of growth of the function. It provides an upper bound on a function ensuring that the function never grows faster than the upper bound. It measures the worst-case complexity of the algorithm. Calculates the longest amount of time taken for execution. Hence, it is a formal way to express the upper boundary of an algorithm's running time.

Let's analyze the graph below:



Let f(n), g(n) be two functions, where n ∈ Z+,

f(n) = O(g(n)) [since f(n) is of the order of g(n)].

If there exists constants 'c' and 'k' such that:

f(n) ≤ c.g(n) for all n≥k

We can conclude that f(n), at no point after n=k, grows faster than g(n). Here, we are calculating the growth rate of the function. This calculates the worst time complexity of f(n).

O(1) example: int arr=new int[10];

This step in any program will execute in O(1) time.

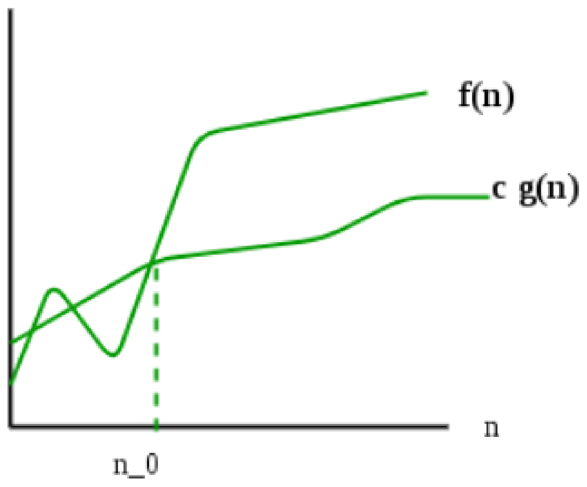
O(n) example: Traversing an array or a linked list. Traversing depends on the size of the array or linked lists.

O(n2) example: Bubble sort algorithm.

2. Omega notation (Ω)

Opposite to big o notation, this asymptotic notation describes the best-case scenario. It is a formal way to present the lower bound of an algorithm running time. This means that this is the minimum time taken for the execution of an algorithm. It is the fastest time that an algorithm can run.

Let's analyze the below graph:



Let f(n), g(n) be two functions, where n ∈ Z+,

f(n) = Ω(g(n)) [since f(n) is of the order of g(n)].

If there exists constants 'c' and 'k' such that:

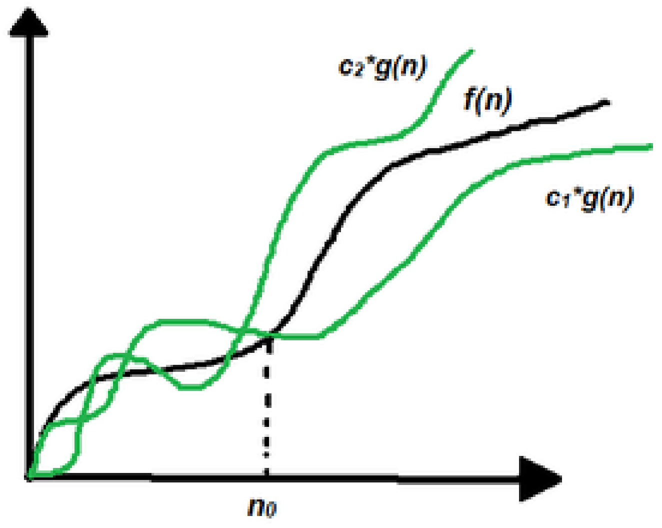
f(n) >= c.g(n) for all n≥k and c>0

As you can observe in the above graph that g(n) is the lower bound of the f(n). Hence, this notation gives us the fastest running time. But we are not interested in the fastest running time. Instead, we are interested in calculating the worst-case scenarios. Because we need to check our algorithm for larger input and what is the worst time that it will take to execute.

3. Theta notation (θ)

This asymptotic notation describes the average case scenario. An algorithm cannot perform worst or best especially when the problem is a real-world problem. The time complexity fluctuates between best case and worst case and this is given by theta notation (θ) which describes the average case.

Theta notation is mainly used when the value of the worst-case and best-case are the same. It represents both the upper bound and the lower bound of the running time for an algorithm. The graphical representation of theta notation is:



Let f(n), g(n) be two functions, where n ∈ Z+,

f(n) = θ(g(n)) [since f(n) is of the order of g(n)].

If there exists constants 'c' and 'k' such that:

c1.g(n) <= f(n) <= c2.g(n)

Here f(n) has a limitation of an upper bound and a lower bound. The condition f(n)= θg(n) will be true only if it satisfies the above equation.

For example:

Assuming we have to reverse the string: “DATAFLAIR”

Let’s have a look at the traditional Algorithm for it:

Step 1: Start

Step 2: Declare two variables str1 and str2.

Step 3: Initialize the variable str1 with “DATAFLAIR”

Step 4: Go to the last character of str1

Step 5: Take each character starting from the end, from str1, and append it to str2.

Step 6: Print str2

Step 7: Stop

In the above algorithm, each time step 5 is executed, a new string is created. It does not append it to the original string. Instead, it creates a new string and assigns it to str2 every time. Thus, the complexity for this algorithm is O(n2).

But in JAVA, we have the StringBuffer.reverse() function which reserves room for characters without reallocation. Therefore, reversing a string using StringBuffer.reverse() in JAVA is possible only in O(1).

Generally, we have three different notations. These notations can be described as follows;

Let’s try to find out the complexity of the linear search algorithm. Suppose you want to search an element in an array using the linear search algorithm. In a linear search algorithm, the search element is compared to each element of the array.

If the element is found at the first position itself, it is a best-case and the program will take the least time for execution. The complexity will be Ω(1). If the element is found at the last position, it will be the worst-case scenario and the program will take maximum time for execution. Its time complexity will be O(n).

The average case complexity is between worst case and best case so it becomes θ(n/1). Since we can ignore the constant terms in asymptotic notations the average-case complexity will be θ(n).

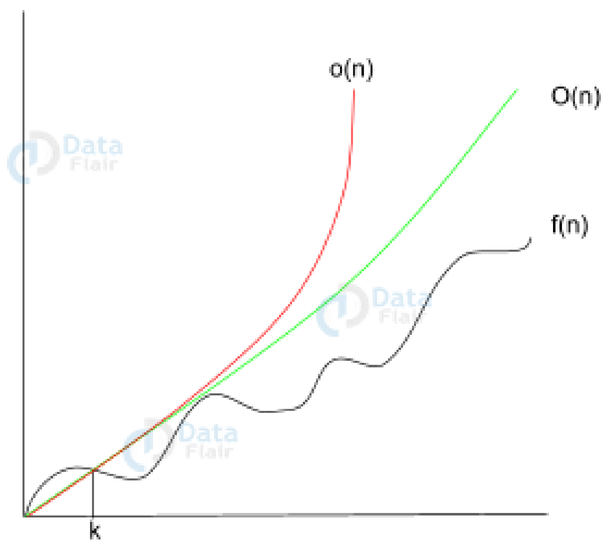
Other Asymptotic Notations

There are other asymptotic notations that are used in data structures as follows;

4. Little 'o' notation

This notation provides a loose upper bound on a function. Little order is an estimate of the order of growth, unlike big O, which is an actual order of growth.

Let's analyze the graph below:



Let f(n), g(n) be two functions, where n ∈ Z+,

f(n) = o(g(n)) [since f(n) is of the order of g(n)].

If for any constant 'c' > 0, there exists a constant 'k' >1, such that:

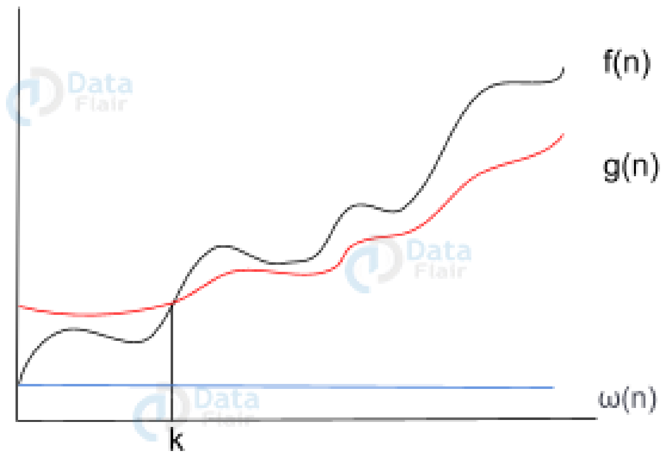
$0 \leq f(n) \leq c.g(n)$.

We can conclude that f(n), at no point after n=k, grows faster than g(n). Here, we are calculating the approximate growth rate of the function.

5. Little ω notation

This notation provides a loose lower bound on a function. Little order is an estimate of the order of growth, unlike big Ω, which is the actual order of growth.

Let's analyze the graph:



Let f(n), g(n) be two functions, where n ∈ Z+,

f(n) = **ω**(g(n)) [since f(n) is of the order of g(n)].

If for any constant 'c' > 0, there exists a constant 'k' >=1, such that:

f(n) > c.g(n) >=0, for every integer n>=k.

Advantages of Asymptotic Notations

1. Asymptotic analysis is the best and efficient way of analyzing algorithms with actual runtime inputs.
2. Analyzing algorithms manually is not feasible as the performance of the algorithm changes with the input change.

Also, the algorithm's performance changes with different machines. Therefore, having a mathematical representation that gives us the actual insight of the maximum and minimum time taken for execution is better.

3. We can represent the upper bound or lower bound of execution time in the form of mathematical equations.
4. Asymptotic analysis of algorithms helps us in performing our task with the best efficiency and fewer efforts.

Commonly used Asymptotic Notations

TYPE	BIG O NOTATION
CONSTANT	O(1)
LINEAR	O(n)
LOGARITHMIC	O(log n)
N LOG N	O(n log(n))
EXPONENTIAL	2O(n)
CUBIC	O(n3)
POLYNOMIAL	nO(1)
QUADRATIC	O(n2)

Example; What is the time complexity of the following program

```
void String(int n){
      if (n<=1) return;
      int i, j;
      for(i=1; i<=n; i++)
            for(j=1; j<=n; j++){
                  prinf(Hello\n”);
                  break;
            }
}
```
	```
	The two for loops can be
	minimized to
	      for(i=1; i<=n; i++)
	            prinf(Hello\n”);
	```

The time complexity is O(n)

The Time Complexity of an algorithm/code is **not** equal to the actual time required to execute a particular code, but the number of times a statement executes.

For example: Write code in C/C++ or any other language to find the maximum between N numbers, where N varies from 10, 100, 1000, and 10000.

You will get surprising results i.e.:

- For N = 10: you may get 0.5 ms time,

- For N = 10,000: you may get 0.2 ms time.
- Also, you will get different timings on different machines. Even if you will not get the same timings on the same machine for the same code, the reason behind that is the current network load.

So, we can say that the **actual time required to execute code is machine-dependent** (whether you are using Pentium 1 or Pentium 5) and also it considers network load if your machine is in LAN/WAN.

What is meant by the Time Complexity of an Algorithm?

Now, the question arises if time complexity is not the actual time required to execute the code, then what is it?

The answer is:

Instead of measuring actual time required in executing each statement in the code, ***Time Complexity considers how many times each statement executes.***

Example 1

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World";
    return 0;
}
```

Output

```
Hello World
```

Time Complexity: In the above code "Hello World" is printed only once on the screen.
So, the time complexity is **constant: O(1)** i.e. every time a constant amount of time is required to execute code, no matter which operating system or which machine configurations you are using.
Auxiliary Space: O(1)

Example2:

```cpp
#include <iostream>
using namespace std;

int main()
{

    int i, n = 8;
    for (i = 1; i <= n; i++) {
        cout << "Hello World !!!\n";
    }
    return 0;
}
```

Output

```
Hello World !!!

Hello World !!!

Hello World !!!

Hello World !!!

Hello World !!!

Hello World !!!

Hello World !!!

Hello World !!!
```

Time Complexity: In the above code "Hello World !!!" is printed only **n times** on the screen, as the value of n can change.

So, the time complexity is **linear: O(n)** i.e. every time, a linear amount of time is required to execute code.

Auxiliary Space: O(1)

Example 3:

```cpp
#include <iostream>
using namespace std;

int main()
{

    int i, n = 8;
    for (i = 1; i <= n; i=i*2) {
        cout << "Hello World !!!\n";
    }
    return 0;
}
```

Output

```
Hello World !!!

Hello World !!!

Hello World !!!

Hello World !!!
```

Time Complexity: O(log2(n))

Auxiliary Space: O(1)

Example 4:

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main()
{

    int i, n = 8;
    for (i = 2; i <= n; i=pow(i,2)) {
        cout << "Hello World !!!\n";
    }
    return 0;
}
```

Output

```
Hello World !!!

Hello World !!!
```

Time Complexity: O(log(logn))

Auxiliary Space: O(1)

How To Find The Time Complexity Of An Algorithm?

Now let us see some other examples and the process to find the time complexity of an algorithm:

Example: Let us consider a model machine that has the following specifications:

- Single processor
- 32 bit
- Sequential execution
- 1 unit time for arithmetic and logical operations
- 1 unit time for assignment and return statements

Q1. Find the Sum of 2 numbers on the above machine:

For any machine, the pseudocode to add two numbers will be something like this:

```
// Pseudocode : Sum(a, b) { return a + b }
#include <iostream>
using namespace std;

int sum(int a,int b)
{
 return a+b;
}

int main() {
      int a = 5, b = 6;
    cout<<sum(a,b)<<endl;
    return 0;
}
```

Output

```
11
```

Time Complexity:

- The above code will take 2 units of time(constant):
    - one for arithmetic operations and
    - one for return. (as per the above conventions).
- Therefore total cost to perform sum operation (**Tsum**) = 1 + 1 = 2
- **Time Complexity = O(2) = O(1)**, since 2 is constant

Auxiliary Space: O(1)

Q2. Find the sum of all elements of a list/array

The pseudocode to do so can be given as:

```
#include <iostream>
using namespace std;

int list_Sum(int A[], int n)

// A->array and
// n->number of elements in array
{
    int sum = 0;
    for (int i = 0; i <= n - 1; i++) {
        sum = sum + A[i];
    }
    return sum;
}

int main()
{
    int A[] = { 5, 6, 1, 2 };
    int n = sizeof(A) / sizeof(A[0]);
    cout << list_Sum(A, n);
    return 0;
}
```

Output

```
14
```

To understand the time complexity of the above code, let's see how much time each statement will take:

```
int list_Sum(int A[], int n)
{
    int sum = 0;    // cost=1  no of times=1
    for(int i=0; i<n; i++)    // cost=2  no of times=n+1 (+1 for the end 
false condition)
        sum = sum + A[i]  ;     // cost=2  no of times=n
    return sum ;                // cost=1  no of times=1
}
```

Therefore the total cost to perform sum operation

*Tsum=1 + 2 * (n+1) + 2 * n + 1 = 4n + 4 =C1 * n + C2 = O(n)*

Therefore, the time complexity of the above code is **O(n)**

Q3. Find the sum of all elements of a matrix

For this one, the complexity is a polynomial equation (quadratic equation for a square matrix)

- Matrix of size n*n => **Tsum = a.n2 + b.n + c**
- Since Tsum is in order of n2, therefore **Time Complexity = O(n2)**

```
#include <iostream>
using namespace std;

int main()
{
    int n = 3;
    int m = 3;
    int arr[][3]
        = { { 3, 2, 7 }, { 2, 6, 8 }, { 5, 1, 9 } };
    int sum = 0;

    // Iterating over all 1-D arrays in 2-D array
    for (int i = 0; i < n; i++) {

        // Printing all elements in ith 1-D array
        for (int j = 0; j < m; j++) {

            // Printing jth element of ith row
            sum += arr[i][j];
        }
    }
    cout << sum << endl;
    return 0;
}
```

Output

```
43
```

Time Complexity: O(n*m)

The program iterates through all the elements in the 2D array using two nested loops. The outer loop iterates n times and the inner loop iterates m times for each iteration of the outer loop. Therefore, the time complexity of the program is O(n*m).

Auxiliary Space: O(n*m)

The program uses a fixed amount of auxiliary space to store the 2D array and a few integer variables. The space required for the 2D array is nm integers. The program also uses a single integer variable to store the sum of the elements. Therefore, the auxiliary space complexity of the program is O(nm + 1), which simplifies to O(n*m).

In conclusion, the time complexity of the program is O(nm), and the auxiliary space complexity is also O(nm).

Fundamental Computing Algorithms

Searching Methods

Introduction

Information retrieval is one of the most important applications of computers. It usually involves giving a piece of information called the key, and ask to find a record that contains other associated information. This is achieved by first going through the list to find if the given key exists or not, a process called searching. Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. The process of searching for an item in a data structure can be quit straightforward or very complex. Searching can be done on internal data structures or on external data structures. Information retrieval in the required format is the central activity in all computer applications. This involves searching. This block deals with searching techniques.

Searching methods are designed to take advantage of the file organisation and optimize the search for a particular record or to establish its absence.

The file organisation and searching method chosen can make a substantial difference to an application's performance.

Basics Searching Techniques

Consider a list of **n** elements or can represent a file of **n records**, where each element is a key / number. The task is to find a particular key in the list in the shortest possible time.

If you know you are going to search for an item in a set, you will need to think carefully about what type of data structure you will use for that set. At low level, the only searches that get mentioned are for sorted and unsorted arrays. However, these are not the only data types that are useful for searching.

❑ **Linear search:** Start at the beginning of the list and check every element of the list. Very slow [order O(n)] but works on an unsorted list.

❑ **Binary Search:** This is used for searching in a sorted array. Test the middle element of the array. If it is too big. Repeat the process in the left half of the array, and the right half if it‟s too small. In this way, the amount of space that needs to be searched is halved every time, so the time is **O(log n)**

❑ **Hash Search:** Searching a hash table is easy and extremely fast, just find the hash value for the item you‟re looking for then go to that index and start searching the array until you find what you are looking for or you hit a blank spot. The order is pretty close to o(1), depending on how full your hash table is.

❑ **Binary Tree search:** Search a binary tree is just as easy as searching a hash table, but it is usually slower (especially if the tree is badly unbalanced). Just start at the root. Then go down the left subtree if the root is too big and the right subtree if is too small. Repeat until you find what you want or the subtree you want isn‟t there. The running time is O(log n) on average and O(n) in the worst case.

Algorithmic Notation

let‟s examine how long it will take to find an item matching a key in the collections. We are interested in:

1. The average time
2. The worst-case time and
3. The best possible time.

However, we will generally be most concerned with the worst-case time as calculations based on worst-case time can lead to guaranteed performance predictions. Conveniently, the worst-case time are generally easier to calculate than average time.

If there are n items in our collection whether it is stored as an array or as linked list-then it is obvious that in the worst case, when there is no item in the collection with the desired key, then n comparisons of the key with keys of the items in the collection will have to be made.

To simplify analysis and comparison of algorithms, we look for a dominated operation and count the number of times that dominant operation has to be performed. In the case of searching, the dominant operation is the comparison, since the search requires n comparisons in the worst case, we say this is **O(n) (pronounce this "big-Oh-n" or "Oh-n")** algorithm.

The best case-in which the first comparison returns a match-requires a single comparison and is **O(1)**.

The average time depends on the probability that the key will be found in the collection-this is something that we would not expected to know in the majority of cases. Thus, in this case, as in most others, estimation of the average time is of little utility. If the performance of the system is vital, i.e. it‟s part of a life-critical system, then we must use the worst case in our design calculations as it represents the best guaranteed performance.

We will now discuss two searching methods and analyze their performance.

These two methods are:

- The sequential search
- The binary search

Sequential Search [Linear search]

This is the most natural searching method. Simply put it means to go through a list or a file till the required record is found. It makes no demands on the ordering of records. The algorithm for a sequential search procedure is now presented.

Algorithm: Sequential Search

This represents the algorithm to search a list of values of to find the required one.

```
INPUT: List of size N. Target value T
OUTPUT: Position of T in the list –I
BEGIN
1. Set FOUND to false
   Set I to 0
2. While (I<=N) and (FOUND is false)
       If List [I] = T
           FOUND = true
         Else
           I=I+1
```

```
    END
3. If FOUND is false
        T is not present in List.      END
```

This algorithm can easily be extended for searching for a record with a matching key value.

Analysis of Sequential Search

Whether the sequential search is carried out on lists implemented as arrays or linked lists or on files, the criterial part in performance is the comparison loop step 2. Obviously the fewer the number of comparisons, the sooner the algorithm will terminate.

The fewest possible comparisons = 1. When the required item is the first item in the list. The maximum comparisons = N when the required item is the last item in the list. Thus if the required item is in position I in the list, I comparisons are required.

Hence the average number of comparisons done by sequential search is

$$\frac{1+2+3.....+N}{N}$$

$$= \frac{-N(N+1)}{2*N}$$

$$= (N + 1)/2$$

Sequential search is easy to write and efficient for short lists. It does not require sorted data. However it is disastrous for long lists. There is no way of quickly establishing that the required item is not in the list or of finding all occurrences of a required item at one place.

We can overcome these deficiencies with the next searching method namely the Binary search.

Example : Program to search for an item using linear search.

```
#include<stdio.h>
/* Search for key in the table */ int seq_search(int key, int a[],
int n)
{
    Int I;      for(i=0;i<n;i++)
      {
            If(a[i]==key) return i+1
          }           return 0;
}
void main()
{
    int I,n,key,pos,a[20];
printf(“Enter the value of n\n”);
 scanf(“%d”,&n);
 printf(“Enter n values\n”;
for(i=0;i<n;i++)
scanf(%d”,&a[i]);
printf(“Enter the item to be searched\n”);
scanf(“%d”, &key);
pos= seq_search(key,n,a);
if(pos==0)
           printf(“Search unscccessful \n”);
 else
            printf(“key found at position = %d \n”,pos);
}
```

Binary Search

The drawbacks of sequential search can be eliminated if it becomes possible to eliminate large portions of the list from consideration in subsequent iterations. The binary search method just that, it halves the size of the list to search in each iteration.

Binary search can be explained simply by the analogy of searching for a page in a book. Suppose you were searching for page 90 in book of 150 pages. You would first open it at random towards the later half of the book. If the page is less than 90, you would open at a page to the right, it is greater than 90 you would open at a page to the left, repeating the process till page 90 was found. As you can see, by the first instinctive search, you dramatically reduced the number of pages to search.

Binary search requires sorted data to operate on since the data may not be contiguous like the pages of a book. We cannot guess which quarter of the data the required item may be in. So we divide the list in the centre each time.

We will first illustrate binary search with an example before going on to formulate the algorithm and analysing it.

Example: Use the binary search method to find 'Scorpio' in the following list of 11 zodiac signs.

Aries 1	Comparison 1 (Leo Scorpio)
Aquarius 2	
Cancer 3	Comparison 2 (Sagittarius Scorpio)
Capricorn 4	Comparison 3 (=scorpio)
Gemini 5	
Leo 6	
Libra 7	
Pisces 8	
Sagittarius 9	
Scorpio 10	
Taurus 11	

This is a sorted list of size 11. The first comparison is with the middle element number 6 i.e. Leo. This eliminates the first 5 elements. The second comparison is with the middle element from 7 to 11, i.e. 9 Sagittarius. This eliminates 7 to 9. The third comparison is with the middle element from 9 to 11, i.e. 10 Scorpio. Thus we have found the target in 3 comparisons. Sequential search would be taken 10 comparisons. We will now formulate the algorithm for binary search.

Algorithm Binary Search
This represents the binary search method to find a required item in a list sorted in increasing order .

INPUT: Sorted LIST of size N, Target Value T

OUTPUT: Position of T in the LIST = I

```
 BEGIN
1.  MAX = N
      MIN = 1
      FOUND = false
2.  WHILE (FOUND is false) and (MAX > = MIN)
        2.1  MID = (MAX + MIN)DIV 2
        2.2  If T = LIST [MID]
                 I=MID
                 FOUND = true
            Else If T < LIST[MID]
                MAX = MID-1
            Else
                MIN = MD+1
        END
```

It is recommended that the student apply this algorithm to some examples.

Analysis of Binary Search:
In general, the binary search method needs no; more than [Iog$_2$n] + 1 comparisons. This implies that for an array of a million entries, only about twenty comparisons will be needed. Contrast this with the case of sequential

search which on the average will need $\frac{(n+1)}{2}$ comparisons.

The conditions (MAX = MIN) is necessary to ensure that step 2 terminates even in the case that the required element is not present. Consider the example of Zodiac signs. Suppose the l0th item was Solar (an imaginary

Zodiac sign). Then at that point we would have

MID = 10

MAX =11

MIN = 9

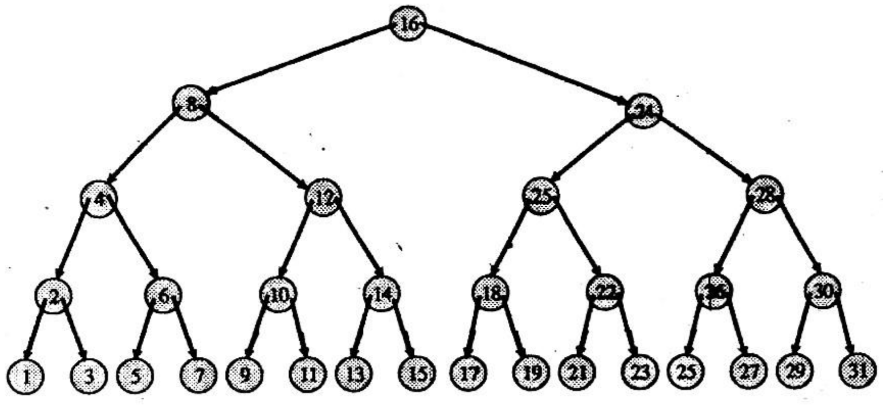
And from 2.2 get

MAX = MID-l = 9

In the next iteration we get (2.1) MID = (9 + 9) DIV 2 = 9
(2.2) MAX= 9-1 = 8.

Since MAX <MIN, the loop terminates. Since FOUND is false, we consider the target was not found.

In the binary search method just described above, it is always the key in the middle of the list currently being examined that is used for comparison. The splitting of the list can be illustrated through a binary decision tree in which the value of a node is the index of the key being tested. Suppose there are 31 records, then the first key compared is at location 16 of the list since (1 + 31)/2 = 16. If the key is less than the key at location 16 the location 8 is tested since (1 + 15)/2 = 8; or if key is less than the key at location 16, then the location 24 is tested. The binary tree describing this process is shown below Figure.



Searching Process in Binary Search

Illustrations of C Programmes

Example : Program to search for an item using Binary Search.

[interpolation search]
#include<stdio.h> int search(item, a,low, high) int

item; /* Element to search */

```
int a[];      /* Element to be searched */
int  low;    /*  Points to the first element */
int  high;  /* Point tot the last element  */
{
 int   mid;           /* Point to the middle element of the table */
if(low>high)      /* No item found  */
return -1;         mid= low+(high-low) *
((item-a[low])/(a[high]-a[low]));
return(item==a[mid]?mid+1:    /* return the middle element */
item<a[mid]?
        search(item,a,low,mid-1):  /* search left part */                                        search(item,a,mid+1,high))
/* search right part */
}
void main()
{           int n, i,a[20],item,pos;
printf("enter the number of elements \n");
 scanf("%d",&n);
printf("Enter %d items \n",n);
for(i=0;i<n;i++)
            {
                scanf("%d",&a[i]);
             }
          printf("Enter the item to be searched \n");
scanf("%d",&item);
pos=search(item,a,0,n-1);    /* 0- low index and n-1 is the high index */
if(pos== -1)
            printf("Item not found \n");
                     else
            printf("Item found at %d position  \n",pos);
```

Sorting Methods

Introduction

Retrieval of information is made easier when it is stored in some predefined order. Sorting is, therefore, a very important computer application activity. Many sorting algorithms are available. Differing environments require differing sorting methods. Sorting algorithms can be characterized in the following two ways:

1. Simple algorithms which require the order of n^2 (written as O (n^2) comparisons to sort n items.
2. Sophisticated algorithms that require the $O(n\log_2 n)$ comparisons to sort items.

The difference lies in the fact that the first method move data only over small distances in the process of sorting, whereas the second method method large distances, so that items settle into the proper order sooner, thus resulting in fewer comparisons. Performance of a sorting algorithm can also depend on the degree of order a heady present in the data.

There are **two basic categories of sorting methods: „Internal Sorting‟** and **„External Sorting‟**. Internal sorting is applied when the entire collection of data to be sorted is small enough that the sorting can take place within main memory. The time required to read or write is not considered to be significant in evaluating the performance of internal sorting methods. External sorting methods are applied to larger collection of data which reside on secondary devices read and write access time are major concern in determining sort performances.

In this unit we will study some methods of internal sorting and External sorting.

Overview of Sorting Methods

In the last units array, the efficient routine for finding data in an array was based on the premise that the data being searched was already sorted. Indeed, computers are used so extensively to process data collections that in many installations, a great deal of their time is spent maintaining that data in sorted order in the first place. It turns out that methods of searching a sorted list have much in common with methods of achieving that sorted condition.

In order to concentrate on sorting abstractions without having to be concerned with the type of data being sorted, most of the code presented in the rest of this unit will be designed to sort only a single kind of data, namely arrays of cardinals. Only minor modifications in a few places are necessary to use the code presented in the next few sections for other kinds of data. There is a surprisingly diverse collection of algorithms that have been developed to solve the apparently simple problem of "Sorting".

The general sorting problem is simple enough to describe: Given an initially unordered array of N records, with one field distinguished as the key, rearrange the records so they are sorted into increasing (or decreasing) order according to each record's key.

Sorting is the problem of taking an arbitrary permutation of n items and rearranging them into the total order.

X_1 □ Xj□ I □ j

- ❑ **Increasing or Decreasing Order?** The same algorithm can be used by both all we need do is change □ to □ in the comparison function as we desire.
- ❑ **What about equal keys?** Does the order matter or not? May be we need to sort on secondary keys or leave in the same order as the original permutations.
- ❑ **What about non-numerical data?** Alphabetizing is sorting text strings and libraries have very complicated rules concerning punctuation etc. Is Brown-Williams before or after Brown America before or after Brown John?

Sorting algorithms are used in all kinds of applications and are necessary for instance, if we plan to use efficient searching algorithms like Binary Search or Interpolation Search since these require their data to be sorted.

There are dozens of algorithms, the choice of which depends on factors such as the number of items relative to working memory, knowledge of the orderliness of the items or the range of the keys, the cost of comparing keys vs. the cost of moving items, etc.

To choose an algorithm we attempt to characterize the performance of the algorithm with respect to an array of size N. We then determine which operations are critical for each type of problem. For example, in sorting we can characterize the performance of a sorting algorithm by.

1. the number of times it compares an element in the array to another value (comparisons) or

2. the number of times it moves an element from or to a position in the array (swaps).

Sorting arrays lets us pay particular attention to the mechanics of the sort algorithm. Arrays, though simple, have the benefits of being internal (memory resident), randomly accessible (using indices), and capable of allowing us to accomplish the sort 'in place'. Of course there are sorts for trees, as well as special sorts for external data structures. But these often observe the sort itself with lots of housekeeping overhead.

The amount of extra memory used by a sort is important. Extra memory is used by sort in the following ways:

- sort in place using no extra memory except for a small stack or table
- use a linked-list and each element requires a link pointer
- need enough extra memory to store a copy of the array to be sorted

Normally, when considering a sorting problem, we will assume that the number of records to be sorted Is small enough that we can fit the entire data set in the computer's memory (RAM) all at once. When this Is true, we can make use of an internal sorting algorithm, which assumes that any key or record can be accessed or moved at any time. That is, we have "random access" to the data.

Sometimes, when sorting an extremely large data set such as Census Data, there are simply , too many records for them to all fit in memory at once. In this case, we have to resort to external sorting algorithms that don't assume we have random access to the data. Instead, these algorithms assume the data is stored on magnetic tapes or disks and only portions of the data will fit in memory. These algorithms use "sequential access" to the data and proceed by reading in, processing, and writing out blocks of records at a time. These partially sorted blocks need to be combined or merged in some manner to eventually sort the entire list.

One final Issue to keep in mind when Implementing a sorting algorithm is the size of the records themselves. Many sorting algorithms move and interchange records in memory several times before they are moved into their final sorted position, For large records, this can add up to lots of execution time spent simply copying data. A popular solution to this problem is called "indirect sorting". The Idea is to sort the indices of the records, rather than the records themselves.

How do you sort?

There are several different ideas which lead to sorting algorithms:

1. Insertion - putting an element in the appropriate place in a sorted list yields a larger sorted list.
2. Exchange - rearrange pairs of elements which are out of order, until no such pairs remain.
3. Selection - extract the largest element form the list, remove it, and repeat.
4. Distribution - separate into piles based on the first letter, then sort each pile.
5. Merging -Two sorted lists can be easily combined to form a sorted list.

There are many different methods used for sorting. Quite frequently, a combination of these methods are used to perform a sort. We will cover four common sorting methods, which we termed selection, insertion, comparison, and divide and conquer. These common sorting methods can be represented as algorithmic functions, which are step by step, problem solving procedures that have a finite number of steps.

Sorting methods can be grouped in to various subgroups that share common themes.
1) Priority queue sorting methods: Example: Selection Sort and
Heap Sort 2) Divided-and-conquer method: Example: MergeSort
and Quicksort 3) Insertion based sort:

Example: InsertionSort 4) Other methods:
Example: BubbleSorl and ShellSort

Evaluating a Sorting Algorithms
There are several performance criteria to be used in evaluating a sorting algorithm:

1. **Running time:** Typically, an elementary sorting algorithm requires **O(N^2)** steps to sort N randomly arranged Items. More sophisticated sorting algorithms require **O(N log** N) steps on average. Algorithms differ in the constant that appears in front of the N^2 or N **log** N. Furthermore, some sorting algorithms are more sensitive to the nature of the input than others. Quicksort, for example, requires **O(N log** N) time in the average case, but requires **O(N^2)** time in the worst case.
2. **Memory requirements:** The amount of extra memory required by a sorting algorithm is also an important consideration. In place sorting algorithms are the most memory efficient since they require practically no additional memory. Linked list representations require an additional N words of memory for a list of pointers. Still other algorithms require sufficient memory for another copy of the input array. These are the most inefficient in terms of memory usage.

3. **Stability:** This is the ability of a sorting algorithm to preserve the relative order of equal keys in a file.

Stability on Sorting algorithm

When a sorting algorithm is applied to a set of records, some of which share the same key, there are several different orderings that are all correctly sorted. If the ordering of records with identical keys is always the same as in the original input, then we say that the sorting algorithm used is "stable". This property can be useful. For instance, consider sorting a list of student records alphabetically by name, and then sorting the list again, but this time by letter grade in a 1 particular course. If the sorting algorithm is stable, then all the students who got "A" will be listed alphabetically. Stability is a difficult property to achieve if we also want our sorting algorithm to be efficient Sorting algorithms are often subdivided into "elementary" algorithms that are simple to implement compared to more complex algorithms that, while more efficient, are also more, difficult to understand, implement, and debug.

It is not always true that the more complex algorithms are the preferred ones, Elementary algorithms are generally more appropriate in the following situations:

1) less than 100 values to be sorted
2) the values will be sorted just once
3) special cases such as:
   a) the input data are "almost sorted"
   b) there are many equal keys.

In general, elementary sorting methods require O(N^2) steps for N random key values. The more complex methods can often sort the same data in just O(N log N) steps. Although it is rather difficult to prove, it can be shown that roughly N log N comparisons are required, in the general case.

Examples of elementary sorting algorithms are: selection sort, insertion sort, shell sort and bubble sort. Examples of sophisticated sorting algorithms are quicksort, radix sort, heapsort and mergesort.

Internal Sorting

In internal sorting, all the data to be sorted is available in the high speed main memory of the computer. We will study the following methods of internal sorting:

1. Insertion sort
2. Bubble sort

3. Quick sort
4. 2-Way Merge sort
5. Heap sort

Insertion Sort

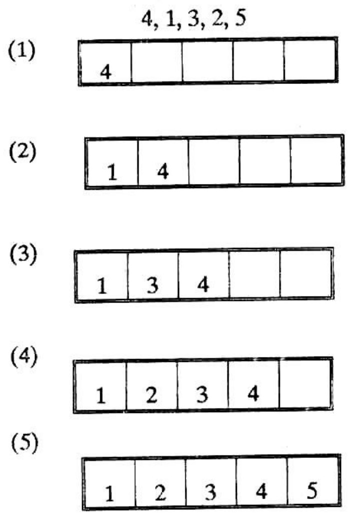
An insertion sort has the advantage that it is simple to understand and simple to implement. Unfortunately, it is rather slow. Given an unsorted array of integer values, an insertion sort visits each element of the array, in turn. As it visits a particular element, it scans the array from the beginning up to the determines where in that segment of the array the current value belongs. It then inserts the current value in that location and shifts every element of the array to the right, up to the present location. It then goes on to the next location (value) in the array. Notice that one index is going from 0 to n and for each such value and another index is scanning the array from 0 to the value of the first index. The result of this is - that this type of sort is $O(n^2)$.

Insertion Sort

This is a naturally occurring sorting method exemplified by a card player arranging the cards dealt to him. He picks up the cards as they are dealt and inserts them into the required position. Thus at every step, we insert an item into its proper place in an already ordered list.

We will illustrate insertion sort with an example below given and presenting the formal algorithm.

Example 1: Sort the following list using the insertion sort method:



Step 1   1 < 4, Therefore insert before 4

Step 2	3 > 1, 3 Insert between 1 & 4
Step 3	2 > I, 2, Insert between I & 3
Step 4	5 > I, 2,3,4, Insert after 4 (5)

Insertion sort

Thus to find the correct position search the list till an item just greater than the target is found. Shift all the items from this point one, down the list.

Insert the target in the vacated slot.

We now present the algorithm for insertion sort.

ALGORITHM: INSERT SORT

INPUT: LIST[ ] of N items in random order.

OUTPUT: LIST[ ] of N items in sorted order.

```
1.  BEGIN
2.  FORI = 2 TO N DO
3.  BEGIN
4.  IF LIST[I] LIST[I-1]
5.  THEN BEGIN
6.  J  = I.
7.  T = LIST [I]     /*STORE LIST [I] */
8.  REPEAT /* MOVE OTHER ITEMS DOWN THE LIST.
9.  J = J-1
10. LIST [J + 1] = LIST [J];
11. IF J = 1 THEN
12. FOUND = TRUE
13. UNTIL (FOUND = TRUE)
14. LIST [I] = T
15. END
16. END
17. END
```

Example : Program to sort n numbers using insertion sort.

```
#include<stdio.h>
void insertion_sort(int a[], int n)
{      int i, j, item;
for(i=0;i<n;i++)
       {
          /* item to be inserted */

item = a[i];

 /* Try from (i-1)th position */

 j=i-1;
         while(j>=0 && item<a[j])
           {
              A[j+1] = a[j]      /* Move the item to the next position */
j--;                  /* and update the position   */
            }
             A[j+1]=item;      /* appropriate position found and so insert item  */
        }
}
void main()
{
 int i, n,a[20];
 printf(“Enter the no. of elements to sort \n”);

scanf(“%d”, &n);      printf(“Enter n elements \n”);

for(i=0;i<n;i++)

scanf(“%d”,&a[i]); insertion_sort(a,n);

printf(“The sorted array is \n”);
for(i=0;i<n;i++)

 printf("%d\n",a[i]);

}
```

Bubble Sort

In a bubble sort we try to improve on the performance of the two previous sorts by making more exchanges in each pass. Again we will make several passes over the array (n -1 to be exact). For the first pass, we begin at element O and proceed forward to element n- 1. Along the way we compare each element to the element that follows it. If the current element is greater than that in the next location, then they are in the wrong order, and we swap them. In this way, an element early in the array that has a very large value is able to percolate (bubble) upward. In the next pass we do exactly the same thing. But now the last element in the array is guaranteed to be where it belongs. so our pass need only proceed as far as element n -2 of the array. This will move the second largest value in the array into the next to last position. This proceeds with each pass encompassing one less element of the array. The result, aside from a sorted array, is that we make n passes over n elements. This sort method, too, is O(n^2).

The bubble sort can be made to work in the opposite direction. moving the least value to the beginning of the array on each pass. This is sometimes referred to as a stone sort. Though the names tend to get confused, In addition there are some pretty obvious improvements that can made rather readily to the bubble sort. For example, at a certain point in our multiple passes over the array it may be the case that the rest of the array is already sorted. As soon as we make a pass in which no exchanges are made this must be the case. So we can keep a counter of exchanges that take place in each pass and quit immediately if the counter is 0 at the end of a pass. Another variation that is often seen is called a shaker sort. In this you simple do one pass of a bubble sort upward, followed by a downward pass of a stone sort. This doesn't gain you much in efficiency, but it is cute.

In this sorting algorithm, multiple Swapping take place in one pass. Smaller elements move or 'bubble' up to the top of the list, hence the name given to the algorithm.

In this method adjacent members of the list to be sorted are compared. If the item on top is greater than the item immediately below it, they are swapped. This process is carried on till the list is sorted.

The detailed **algorithm follows**:

Algorithm for Bubble Sort

INPUT: LIST [] of N items in random order

O UTPUT: LIST [] of N items sorted in ascending order.

```
1.  SWAP = TRUE
    PASS = 0/
2.  WHILE SWAP = TRUE DO
    BEGIN.
2.1   FOR I = 0 TO (N-PASS) DO
          BEGIN
          2.1.1  IFA[I] >A [I+1]
                 BEGIN
                 TMP = A[I]
                 A[I] = A[I + 1]
                 A[I + 1] = TMP
                 SWAP = TRUE
                 END
                 ELSE
                 SWAP = FALSE
          2.1.2  PASS = PASS + 1
                 END
                 END
```

Total number of comparisons in Bubble sort are

= (N -1) + (N -2) ...+ 2 + 1

$$= \frac{(N-1)* N}{2} \qquad = O(N^2)$$

This inefficiency is due to the fact that an item moves only to the next position in each pass.

Example: C program, Function to arrange numbers in ascending order using bubble sort technique.

```
#include<stdio.h> void bubble_sort(int a[], int n)
{
 int  i;    /* To access subsequent item while comparing*/
int    j;  /* Keep track of the passes  */
int temp;  /* Used to exchange the item */
int   sum; /* Holds the total number of exchanges */
 int  pass; /*Holds the number of passes required */
 int  exchag;  /* Holds the number of exchanges in each pass */
int flag;  /* Indicate any exchange has been done or not */
 sum = 0;        pass = 0;
for(j=1;j<n;j++)
       {
exchg = 0;  /* number of exchanges just before the pass */
flage = 0;   /* No exchange been done */
for(i=0;i<n-j;i++)
             {                    if(a[i]>=a[i+1])
                  {
  /* Exchange and update the number of exchange in the current pass*/
 temp=a[i];
 a[i]=a[i+1];
a[i+1=temp;
exchg++;
sum++  /* Update the total number of exchanges */
 flag=1; /* Exchange has been done */
                  }
             }
```

```
            pass++;    /* update the number of passes */
printf("Number of exchanges in pass  : %d=%d\n",j,exchg);
print("Total number of exchanges = %d\n",sum);
}
void main()
{
int i,n,a[20];
   printf("Enter the number of items to sort\n);
scanf(%d,&n);
   print("Enter the items to sort\n);
for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 bubble_sort(a,n);
 printf("The sorted items are \n");
for(i=0;i<n;i++)
     {
          Printf("%d\n",a[i]);
      } }
```

Note: At least one pass is required to check whether the items are sorted.

So, the best case time complexity is O(1).

Selection Sort

A selection sort is slightly more complicated. but relatively easy to understand and to code. It is one of the slow sorting techniques. Given an unsorted array of integer values, a selection sort visits each element of the array, in turn. As it visits an element, it employs a second index to scan the array from the present location to the end while it identifies the least (smallest) value in that segment of the array. It then swaps that least value with the current value. Because one index scans the array once while another index scans part of the array each time, this sort algorithm is also O(n^2).

Simple steps for selection sort.

[Assume we are sorted the n items in array A]

```
for i = 1 to n do
 for j = j+1 to n do
        if A[i] > A[j]  then swap(A[i],A[j])
```

Example:

Program to illustrate the Selection sort, assume we have an array „A‟ with „n‟ elements.

```
Void selectionsort(int A[], int n)
 {
    Int minindex,j,p,tmp;
for(p=0;p< n-1;p++)
     {
        minindex = p;
for (j= p+1; j<n; j++)
 if(A[j]<A[minindex]
 minindex = j;
       }
       tmp = A[p];
A[p]=A[minindex];
        A[minindex]=tmp;   }
```

Example : C program, Function to arrange number in ascending order using Selection sort technique.

```
#include<stdio.h>
void selection_sort(int a[],int n);
{
int i,j,pos,small,temp;
for(i=0;i<n-1;i++)
         {
small=a[i];                /* Initial small number in ith  pass   */
pos=i;                    /* Position of smaller number */
                /* Find the minimum of remaining elements along with the position */
for(j=i+1;j<n;j++)
                 {                        if(a[i]<small)
                        {                              small=a[j];
pos=j;
                          }
                  }
              /* Exchange ith item with least item */
temp=a[pos];
 a[pos] = a[i];
 a[i]=temp;
          }
}
void main()
{      int i,n,a[20];
printf("Enter the number of  elements to sort\n");
scanf(%d",&n);
printf("Enter %d elements to sort \n",n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
```
selection_sort(a,n)
```
printf("The sort elements are \n");
for(i=0;i<n;i++)
printf("%d",&a[i]);   }
```

Shell Sort

The shell sort is an improved version of the selection sort. It attempts to affect this improvement by finding and making exchanges that have a big impact on the eventual sorted order. That is, we want a value to make one long jump to near its eventual location, rather than lots of little exchanges that move it little closer each time. To do this we select a gap, usually something slightly less than half the size of the array. We then make a pass along the array comparing elements that are a gap distance apart. If they need to be swapped. then we do so. On the next pass and subsequent passes we decrease the size of the gap by half. Thus on our last pass the gap is just one. But by then all values are quite near their final location. This sort can be improved on slightly using the kind of modifications that we have seen in earlier sorts. The best optimized version of shell sort is theoretically O($n^{1.2}$).

Simple steps for Shell sort : [Assume we have an array „A‟ with „n‟ elements] for(gap=n/2; gap>0; gap/=2)

```
 {
   for(i=gap; i<n; i++)
       for(j=i-gap; j>=0; j -= gap)
         {
           if(!COMPARE(a, j, ,j+gap))
 break;
          SWAP(a, j, j+gap);
        }    }
```

Example :

Program to illustrate the Shell sort, assume we have an array „A‟ with „n‟ elements.

```
void shellsort(int A[], int n)
{    int p, j, increment, tmp;
for( increment = 1;increment<=n; increment*=2);
for( increment = (increment/2)-1; increment >0;
increment = ( increment - 1)/2);
        {
          for(p= increment; p<n; p++)
```

```
            {              tmp=A[p];
                for(j=p; j>= increment && A[j- increment]>tmp;
j-= increment)
                  A[j]=A[j- increment]
                 A[j]=tmp;
              }
           }
      }
```

Quick Sort

Quick sort doesn't look at all like any of the sorts we have examined up until now. It is a partition sort. It is one of the speediest of these 'in place' array sorts. It is also one of the most complex to code. We say that it is faster than its siblings, but in fact, it is also an $O(n^2)$ algorithm. It gets its reputation for speed from the fact that it is $O(n^2)$. only in the worst case, which almost never occurs. In practice quick sort is an O(n Log n) algorithm.

Quick sort begins by picking an element to be the pivot value. There is much debate, about how to best pick this initial element. In terms of understanding how quick sort works, it really doesn't matter. They can just pick the first element in the array. With the pivot to work with, the array is divided into three parts: all values less than the pivot, the pivot, and an values greater than the pivot. When this is finished, the pivot value is in its proper position in the sorted array. Quick sort, then, recursively applies this same process to the first partition, containing low values, and to the second position, which contains high values. Each time at least one element (the pivot) finds its final resting place and the partitions get smaller. Eventually the partitions are of size one and the recursion ends.

Quick sort is fast. It is also difficult to code correctly the first time. Most designers seem to believe that implementing quick sort for data set sizes less than 300 to 500 is wasted effort.

This is the most widely used internal sorting algorithm. In its basic form, it was invented by C.A.R. Hoare in 1960. Its popularity lies in the ease of implementation, moderate use of resources and acceptable behaviour for a variety of sorting cases.

The basis of quick sort is the 'divide' and conquer' strategy i.e. Divide the problem [list to be sorted] into sub-problems [sub-Iists], until solved sub problems [sorted of. sub-lists] are found. This is implemented as

1. Choose one item A [I] from the list A[ ].
2. Rearrange the list so that this item is in the proper position i.e. all preceding items have a lesser value and all succeeding items have a greater value than this item.
    1. A [0], A[1]…...A[I-1] in sub list 1
    2. A [I]
    3. A [I + 1], A[I + 2]……...A[N] in sublist 2
3. Repeat steps 1 & 2 for sublist 1 & sublist 2 till A[ ] is a sorted list.

   As can be seen, this algorithm has a recursive structure.

Step 2 or the 'divide' procedure is of at most importance in this algorithm.

This is usually implemented as follows:

1. Choose A[I] the dividing element.
2. From the left end of the list (A[0] onwards) scan till an item A[R] is found whose value is greater than A[l]
3. From the right end of list [A[N] backwards] scan till an item A[L] is found whose value is less than A[I].
4. Swap A[R] & A[L].
5. Continue steps 2, 3 & 4 till the scan pointers cross. Stop at this stage.
6. At this point sublist1 & sublist2 are ready.
7. Now do the same for each of sublist1 & sublist2.

We will now give the implementation of Quicksort and illustrate it by an example.

```
Quicksort (int A[], int X, int I)
{     int L, R, V
1.     If (IX)
      {
```

```
 2.  V= A[I], L = X-I, R = I;
 3.  For (;;)
    {
 4.  While (A[ + + L] V);
 5.  While (A[--R] V);
 6.  If (L = R) /* left & right ptrs. have crossed */
 7.  break;
 8.  Swap (A, L, R) /* Swap A[L] & A[R] */
    }
 9.  Swap (A, L, I)
10.  Quicksort (A, X, L-1)
11.  Quicksort (A, L + 1, I)
      }
       }
```

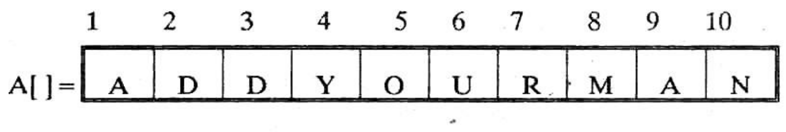
Quicksort is called with A,I, N to sort the whole file.

Example:

Consider the following list to be sorted in ascending order. 'ADD YOUR MAN'.

(Ignore blanks)

N = 10



	1	2	3	4	5	6	7	8	9	10
A[ ]=	A	D	D	Y	O	U	R	M	A	N

A[ ]=

Quicksort (A, 1, 10)

1. l0 >1
2. V= A[10] = 'N'   L =I-1= 0

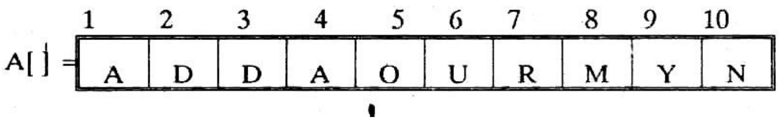
   R = I= 10

4. A [4]="Y">V; therefore, L=4

5.   A [9]=‟A‟ <V; therefore, R=9

6.   L < R

8.   SWAP (A, 4,9) to get



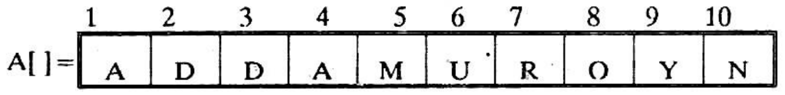
A[ ]	1	2	3	4	5	6	7	8	9	10
	A	D	D	A	O	U	R	M	Y	N

4.   A[5] =‟O‟ > V; Therefore L =5

5.   A[8] -'M' < V; Therefore R =8

6.   L<R

8.   SWAP (A, 5,8) to get]



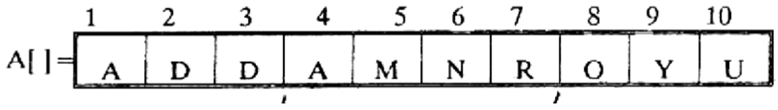
A[ ]	1	2	3	4	5	6	7	8	9	10
	A	D	D	A	M	U	R	O	Y	N

4.   A[6]=‟U‟ >V;..L=6

5.   A[5]=‟M‟<V;R=5

6.   L<R,..break.

9.   SWAP (A,6,10) to get



A[ ]	1	2	3	4	5	6	7	8	9	10
	A	D	D	A	M	N	R	O	Y	U

At this point „N‟ is in its correct place. A[6], A[1] to A[5] constitutes
sublist1.

A[7] to A[10] constitutes sublist2. Now

10.   Quicksort (A,1,5)

11.   Quicksort (A,6,10)

The Quicksort algorithm uses the O(N Log_2 N).comparisons on average.

The performance can be improved by keeping in mind the following points.

1. Switch to a faster sorting scheme like insertion sort when the sublist size becomes comparatively small.

2. Use a better dividing element I in the implementations; We have always used A[N] as the dividing element. A useful method for the selection of a dividing element is the Median-of three method.

Select any 3 elements from the list. Use the median of these as the dividing element.

Example : C program to sort the numbers in ascending order using quick sort.

```
#include<stdio.h>
/* Function to partition the array for quick sort*/
int partition(int a[],int low, int high)
{          int i,j,temp,key;
key=a[low];
 i=low+1;
j=high;
while(1)
            {
                while(i<high&&key>=a[i])
                         i++;
 while(key<a[j])
 j--;
 if(i<j)
 {
 temp=a[i];
 a[i]=a[j];
 a[j]=temp;
                        }
else
{
temp=a[low];
a[low]=a[j];
```

```
 a[j]=temp;
                    }
               }
}

/* function to sort the numbers in ascending order using quick sort */
void quicksort(int a[], int low, int high)
{
int j;
if(low<high)
        {
            j=partition(a,low,high);    /* partion the array into 2 subtables */
 quicksort(a,low,j-1)          /* Sort the left part of the array */
 quicksort(a,j+1,high);      /* Sort the right part of the array */
         }
}

void main()
 {    int I,n,a[20];
   printf("Enter the value for n \n");

scanf(%d",&n);    printf("Enter the number to be sorted \n);
for(i=0;i<n;i++)

 scanf(%d", &a[i]);

 quicksort(a,0,n-1);

printf("The sorted array is \n");

for(i=0;i<n;i++)
```

```
printf("%d\n",a[i]);   }
```

Tree Sort

The use of a tree to sort an array is also a departure from the sorts that we have looked at so far. While all of those earlier sorts did their work 'in place', a tree sort requires additional space for a tree to be constructed. Our goal, in every case, has been to end up with the original array in sorted order. To do that with a tree sort we must copy the data from the tree back into the array by doing an in-order traversal of the completed tree.

A tree sort proceeds by visiting each element of the array and adding it to an ordered binary tree. The obvious disadvantage to this is the additional space required to hold the tree. When all of the elements of the array have been added to the tree, we walk the tree and repopulate the array in sorted order.

In the worst case, the original array is already in sorted order. If this happens, then for each element of the array we will end up adding that element as a leaf on a tree that is really a linked list. In this worst case a tree sort is $O(n^2)$. In practice this seldom happens, so tree sort is presumed to be O(n Log n). This is a good example of a typical space/time trade-off.

Heap Sort

A Heap sort is an efficient sort. It Is also, perhaps, the most difficult to understand. A heap sort can be done 'in place'. That is, it can be done in an array without creating any additional data structure. Recall that a tree sort has to build a tree as it proceeds. A heap sort does not have to create an actual heap (tree) representation. It can simply rearrange the array into a heap representation.

The representation of a heap using an array depends on the fact that a tree representing a heap is a complete tree. There are no gaps between the leaves on the bottom level. The first element of the array is the root of the heap. The second and third elements of the array are level 1 of the heap. The third, fourth, fifth, and sixth elements are level 2 of the heap, and so on. If the heap is not a full tree, there will be some elements missing from the part of the array that represents the final level of the heap; but these will be on the end of the array. There will be no gaps.

A heap sort can be accomplished by heapifying the original, unsorted array. Then select the root of the heap, which must be the largest element. Swap this value with the last element in the heap and reheapify .Notice that the heap portion of the array shrinks as you do this. The array is partitioned into the sorted portion, at the end, and the heap portion, at the beginning. The sorted portion grows and the heap portion shrinks until only the entire array is sorted.

External Sorts

Merge Sort

Like the other sorts we have seen, we will look at merge sort (and later at radix sort) In terms of a simple Internal array. These sorts, though, are really only useful when they are applied to large external data structures. Merge sort is a very old sort. It was used extensively when the primary external data store was magnetic tapes. It actually takes its name from the action of merging 2 or more sorted tapes to make a single sorted tape. In fact, there are a variety of sorts known a merge sorts. They have names such as balanced merge, natural merge, and polyphase merge; and are all variations on the same theme.

A true merge sort is a two part process. The first phase is called the distribution phase and the second is called the merge phase. These two phases may be alternated several times before the sort is complete. In the course of the distribution phase, data elements must be written to a new array (or a new file). As a consequence, merge sorts (all of them) require at least 2n space.

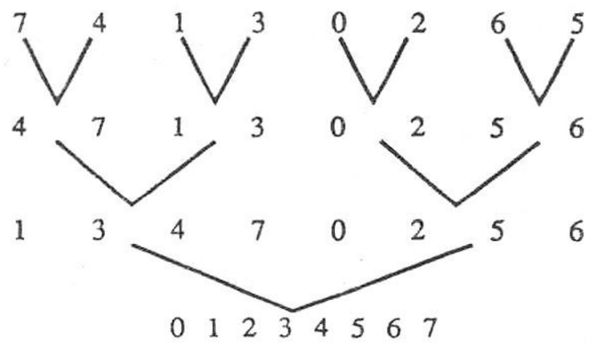
In a distribution phase element from the original (or current) array are written into a new array (or arrays) such that the new array(s) are individually sorted, This can be done in several ways. One way is to select items from the original array and place them into bucket arrays. There might be a bucket to hold values 0-10, another for values 11-20, and so on. The buckets can be sorted independently, then passed to a merge phase to recombine them into a final sorted file.

In another approach to the distribution phase, values are taken from the original array until there is a drop down (i.e. an out of order value is encountered). The new partial

An array is then merged with what remains of the original array. In this fashion, each distribution/merge phase guarantees that one element is sorted. This approach, however, may require up to n distribution/merge phases. Nonetheless, merge sorts are O(n log n), with the drawback that they need extra space to operate.

2-Way Merge Sort

Merge sort is also one of the 'divide and conquer' class of algorithms. The basic idea into this is to divide the list into a number of sublists, sort each of these sublists and merge them to get a single sorted list. The recursive implementation of 2- way merge sort divides the list into 2 sorts the sublists and then merges them to get the sorted list. The illustrative implementation of 2 way merge sort sees the input initially as n lists of size 1. These are merged to get 0/2 lists of size 2. These n/2 lists are merged pair wise and so on till a single list is obtained. This can be better understood by the following example. This is also called CONCATENATE SORT.



2-way merge sort

We give here the recursive implementation of 2 Way Merge Sort

```
Mergesort (int List[ ], int, low, int high)
{    int  mid;
1.  Mid = (low + high)/2;
2.  Mergesort (LIST, low, mid);
3.  Mergesort (LIST, mid + 1, high);
4.  Merge (low, mid, high, List, FINAL)
}
Merge (int low, int mid, int high, int LIST [ ], int FINAL)
{
int a, b, c, d;
 a = low b = low c = mid + 1
While (a < = mid and c < = high) do
   {
        If LIST [ a] < = LIST [c] then
         {
            FINAL [b] = LIST [a]
a=a+1
            }             else
            {
                 FINAL [b] = LIST [c]
c=c+1
            }
b=b+1
   }
If (a > mid) then
for d = c to high do
      {
         B [bl = LIST [d]
b = b+1
       }
Else
for d = a to mid do
      {
B[b] = A[d]
b = b+l.
       }
}
```

To sort the entire list, Mergesort should be called with LIST, 1, N.

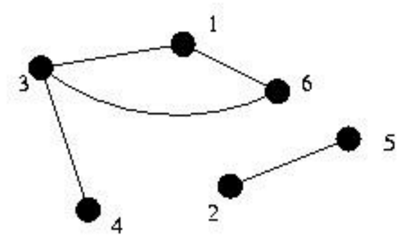
Mergesort is the best method for sorting linked lists in random order. The total computing time is of the 0(0 $\log_2$ n).

The disadvantage of using mergesort is that it requires two arrays of the same size and type for the merge phase. That is, to sort and list of size n, it needs space for 2n elements.

Graphs and their Applications – I

Introduction to Graphs

A ***graph*** is a collection of ***vertices*** V and a collection of ***edges*** E consisting of pairs of vertices. Think of vertices as *locations*. The set of vertices is the set of all the possible locations. In this analogy, edges represent paths between pairs of those locations. The set E contains all the paths between the locations.



Representation

The graph is normally represented using that analogy. Vertices are points or circles, edges are lines between them.

In this example graph:

V = {1, 2, 3, 4, 5, 6}

E = {(1,3), (1,6), (2,5), (3,4), (3,6)}.

Each ***vertex*** is a member of the set V. A vertex is sometimes called a ***node***.

Each ***edge*** is a member of the set E. Note that some vertices might not be the end point of any edge. Such vertices are termed ***isolated***.

Sometimes, numerical values are associated with edges, specifying lengths or costs; such graphs are called ***edge-weighted*** graphs (or ***weighted*** graphs). The value associated with an edge is called the ***weight*** of the edge.

A similar definition holds for node-weighted graphs.

Examples of Graph Problems

Telecommunication

Given a set of computers and a set of wires running between pairs of computers, what is the minimum number of machines whose crash causes two given machines to be unable to communicate? (The two given machines will not crash.)

Graph: The vertices of the graph are the computers. The edges are the wires between the computers. Graph problem: minimum dominating sub-graph.

Sample Problem: Riding The Fences

Farmer John owns a large number of fences, which he must periodically check for integrity. He keeps track of his fences by maintaining a list of points at which fences intersect. He records the name of the point and the one or two fence names that touch that point. Every fence has two end points, each at some intersection point, although the intersection point may be the end point of only one fence.

Given a fence layout, calculate if there is a way for Farmer John to ride his horse to all of his fences without riding along a fence more than once. Farmer John can start and finish anywhere, but cannot cut across his fields (i.e., the only way he can travel between the intersection points is along a fence). If there is a way, find one way.

Graph: Farmer John starts at intersection points and travels between the points along fences. Thus, the vertices of the underlying graph are the intersection points, and the fences represent edges. Graph problem: Traveling Salesman Problem.

Knight moves

Given: Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.

Graph: The graph here is harder to see. Each location on the chessboard represents a vertex. There is an edge between two positions if it is a legal knight move. Graph Problem: Single Source Shortest Path.

Overfencing

Farmer John created a huge maze of fences in a field. He omitted two fence segments on the edges, thus creating two “exits" for the maze. The maze is a “perfect" maze; you can find a way out of the maze from any point inside it.

Given the layout of the maze, calculate the number of steps required to exit the maze from the “worst" point in the maze (the point that is “farther" from either exit when walking optimally to the closest exit). Here's what one particular W=5, H=3 maze looks like:

```
+-+-+-+-+-+
|         |
+-+ +-+ + +
|   ||    |
+ +-+-+ + +
|      |  |
+-+ +-+-+-+
```

Graph: The vertices of the graph are positions in the grid. There is an edge between two vertices if they represent adjacent positions that are not separated by a wall. Graph problem: Shortest Path.

Terminology

An edge is a ***self-loop*** if it is of the form (u,u). The sample graph contains no self-loops.

A graph is ***simple*** if it neither contains self-loops nor contains an edge that is repeated in E. A graph is called a ***multigraph*** if it contains a given edge more than once or contain self-loops. For our discussions, graphs are assumed to be simple. The example graph is a simple graph.

An edge (u,v) is ***incident*** to both vertex u and vertex v. For example, the edge (1,3) is incident to vertex 3.

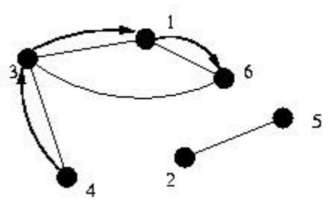
The ***degree*** of a vertex is the number of edges which are incident to it. For example, vertex 3 has degree 3, while vertex 4 has degree 1.

Vertex u is ***adjacent*** to vertex v if there is some edge to which both are incident (that is, there is an edge between them). For example, vertex 2 is adjacent to vertex 5.

A graph is said to be ***sparse*** if the total number of edges is small compared to the total number possible ((N x (N-1))/2) and ***dense*** otherwise. For a given graph, whether it is dense or sparse is not well-defined.

Directed Graph

Graphs described thus far are called ***undirected***, as the edges go `both ways'. So far, the graphs have connoted that if one can travel from vertex 1 to vertex 3, one can also travel from vertex 1 to vertex 3. In other words, (1,3) being in the edge set implies (3,1) is in the edge set. Sometimes, however, a graph is ***directed***, in which case the edges have a direction. In this case, the edges are called ***arcs***. Directed graphs are drawn with arrows to show direction.

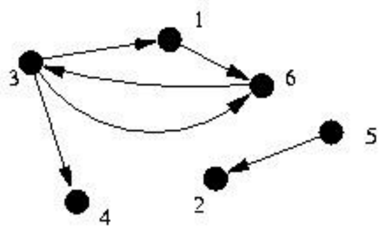


The ***out-degree*** of a vertex is the number of arcs which begin at that vertex. The ***in-degree*** of a vertex is the number of arcs which end at that vertex. For example, vertex 6 has in-degree 2 and out-degree 1.

A graph is assumed to be undirected unless specifically called a directed graph.

Paths

A ***path*** from vertex u to vertex x is a sequence of vertices (v 0, v 1, ..., v k) such that v 0 = u and v k = x and (v 0, v 1) is an edge in the graph, as is (v 1, v 2), (v2, v 3), etc. The length of such a path is k.



For example, in the undirected graph above, (4, 3, 1, 6) is a path. This path is said to ***contain*** the vertices v 0, v 1, etc., as well as the edges (v 0, v 1), (v 1, v 2), etc.

Vertex x is said to be ***reachable*** from vertex u if a path exists from u to x.

A path is ***simple*** if it contains no vertex more than once.

A path is a ***cycle*** if it is a path from some vertex to that same vertex. A cycle is ***simple*** if it contains no vertex more than once, except the start (and end) vertex, which only appears as the first and last vertex in the path.

These definitions extend similarly to directed graphs (e.g., (v 0, v 1), (v 1, v 2), etc. must be arcs).

Graph Representation

The choice of representation of a graph is important, as different representations have very different time and space costs. The vertices are generally tracked by numbering them, so that one can index them just by their number. Thus, the representations focus on how to store the edges.

Edge List

The most obvious way to keep track of the edges is to keep a list of the pairs of vertices representing the edges in the graph. This representation is easy to code, fairly easy to debug, and fairly space efficient. However, determining the edges incident to a given vertex is expensive, as is determining if two vertices are adjacent. Adding an edge is quick, but deleting one is difficult if its location in the list is not known. For weighted graphs, this representation also keeps one more number for each edge, the edge weight. Extending this data structure to handle directed graphs is straightforward. Representing multigraphs is also trivial.

Example

The sample undirected graph might be represented as the following list of edges:

	V1	V2
e1	4	3
e2	1	3
e3	2	5
e4	6	1
e5	3	6

Adjacency Matrix

A second way to represent a graph utilized an *adjacency matrix*. This is a N by N array (N is the number of vertices). The i,j entry contains a 1 if the edge (i,j) is in the graph; otherwise it contains a 0. For an undirected graph, this matrix is symmetric. This representation is easy to code. It's much less space efficient, especially for large, sparse graphs. Debugging is harder, as the matrix is large. Finding all the edges incident to a given vertex is fairly expensive (linear in the number of vertices), but checking if two vertices are adjacent is very quick. Adding and removing edges are also very inexpensive operations.

For weighted graphs, the value of the (i,j) entry is used to store the weight of the edge. For an unweighted multigraph, the (i,j) entry can maintain the number of edges between the vertices. For a weighted multigraph, it's harder to extend this.

Example

The sample undirected graph would be represented by the following adjacency matrix:

	V1	V2	V3	V4	V5	V6
V1	0	0	1	0	0	1
V2	0	0	0	0	1	0
V3	1	0	0	1	0	1
V4	0	0	1	0	0	0
V5	0	1	0	0	0	0
V6	1	0	1	0	0	0

It is sometimes helpful to use the fact that the (i,j) entry of the adjacency matrix raised to the k-th power gives the number of paths from vertex i to vertex j consisting of exactly k edges.

Adjacency List

The third representation of a matrix is to keep track of all the edges incident to a given vertex. This can be done by using an array of length N, where N is the number of vertices. The i-th entry in this array is a list of the edges incident to i-th vertex (edges are represented by the index of the other vertex incident to that edge).

This representation is much more difficult to code, especially if the number of edges incident to each vertex is not bounded, so the lists must be linked lists (or dynamically allocated). Debugging this is difficult, as following linked lists is more difficult. However, this representation uses about as much memory as the edge list. Finding the vertices adjacent to each node is very cheap in this structure, but checking if two vertices are adjacent requires checking all the edges adjacent to one of the vertices. Adding an edge is easy, but deleting an edge is difficult, if the locations of the edge in the appropriate lists are not known. Extend this representation to handle weighted graphs by maintaining both the weight and the other incident vertex for each edge instead of just the other incident vertex. Multigraphs are already representable. Directed graphs are also easily handled by this representation, in one of several ways: store only the edges in one direction, keep a separate list of incoming and outgoing arcs, or denote the direction of each arc in the list.

Example
The adjacency list representation of the example undirected graph is as follows:

Vertex	Adjacent Vertices
1	3, 6
2	5
3	6, 4, 1
4	3
5	2
6	3, 1

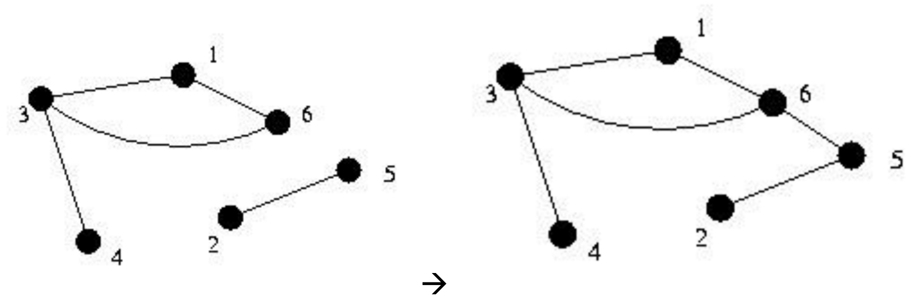
Implicit Representation
For some graphs, the graph itself does not have to be stored at all. For example, for the Knight moves and Overfencing problems, it is easy to calculate the neighbors of a vertex, check adjacency, and determine all the edges without actually storing that information, thus, there is no reason to actually store that information; the graph is implicit in the data itself. If it is possible to store the graph in this format, it is generally the correct thing to do, as it saves a lot on storage and reduces the complexity of your code, making it easy to both write and debug.

If N is the number of vertices, M the number of edges, and d max the maximum degree of a node, the following table summarizes the differences between the representations:

Efficiency	Edge List	Adj Matrix	Adj List
Space	2*M	N^2	2xM
Adjacency Check	M	1	d max

List of Adjacent Vertices	M	N	d max
Add Edge	1	1	1
Delete Edge	M	2	2*d max

Connectedness



An undirected graph is said to be *connected* if there is a path from every vertex to every other vertex. The example graph is not connected, as there is no path from vertex 2 to vertex 4. However, if you add an edge between vertex 5 and vertex 6, then the graph becomes connected.

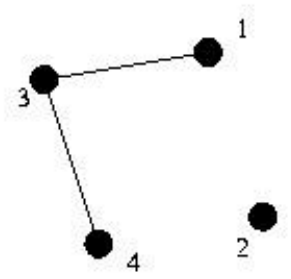
A *component* of a graph is a maximal subset of the vertices such that every vertex is reachable from each other vertex in the component. The original example graph has two components: {1, 3, 4, 6} and {2, 5}. Note that {1, 3, 4} is not a component, as it is not maximal.

A directed graph is said to be *strongly connected* if there is a path from every vertex to every other vertex.

A *strongly connected component* of a directed graph is a vertex u and the collection of all vertices v such that there is a path from u to v and a path from v to u.

Subgraphs

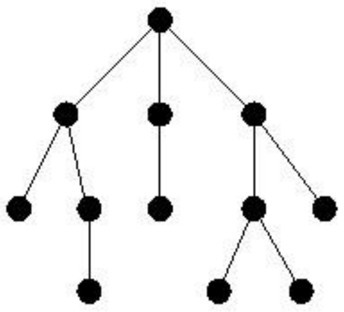
Graph G' = (V', E') is a subgraph of G = (V, E) if V' is a subset of V and E' is a subset of E.



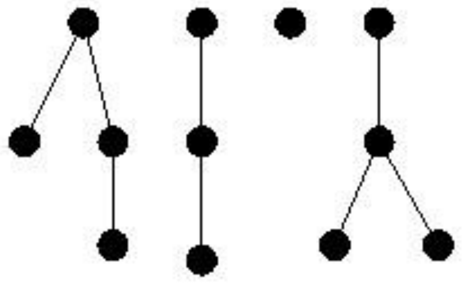
The subgraph of G *induced* by V' is the graph (V', E'), where E' consists of all the edges of E that are between members of V'. For example, for
V' = {1, 3, 4, 2}, the subgraph is like the one shown ->

Special Graphs
An undirected graph is said to be a *tree* if it contains no cycles and is connected.

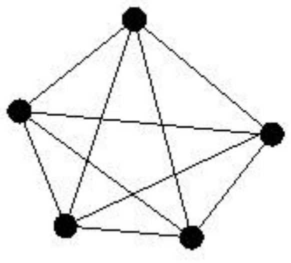
Rooted tree
Many trees are what is called *rooted*, where there is a notion of the "top" node, which is called the root. Thus, each node has one *parent*, which is the adjacent node which is closer to the root, and may have any number of *children*, which are the rest of the nodes adjacent to it. The tree above was drawn as a rooted tree.



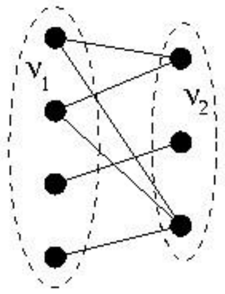
Forest
An undirected graph which contains no cycles is called a *forest*. A directed acyclic graph is often referred to as a *dag*.



Complete Graph
A graph is said to be ***complete*** if there is an edge between every pair of vertices.



Bipartite Graph
A graph is said to be ***bipartite*** if the vertices can be split into two sets V1 and V2 such there are no edges between two vertices of V1 or two vertices of V2.



Uninformed Search
Searching is a process of considering possible sequences of actions, first you have to formulate a goal and then use the goal to formulate a problem.

A **problem** consists of four parts: the **initial state**, a set of **operators**, a **goal test** function, and a **path cost** function. The environment of the problem is represented by a **state space**. A **path** through the state

space from the initial state to a goal state is a **solution**. In real life most problems are ill-defined, but with some analysis, many problems can fit into the state space model. A single general search algorithm can be used to solve any problem; specific variants of the algorithm embody different strategies.
Search algorithms are judged on the basis of **completeness**, **optimality**, **time complexity**, and **space complexity**. Complexity depends on *b*, the **branching factor** in the state space, and *d*, the **depth** of the shallowest solution.

Completeness: is the strategy guaranteed to find a solution when there is one?

Time complexity: how long does it take to find a solution? Space complexity: how much memory does it need to perform the search?

Optimality: does the strategy find the highest-quality solution when there are several different solutions?

This 6 search type below (there are more, but we only show 6 here) classified as uninformed search, this means that the search have no information about the number of steps or the path cost from the current state to the goal - all they can do is distinguish a goal state from a non-goal state. Uninformed search is also sometimes called blind search. The 6 search type are listed below:

Breadth-first search expands the shallowest node in the search tree first. It is complete, optimal for unit-cost operators, and has time and space complexity of O(b^d). The space complexity makes it impractical in most cases. Using BFS strategy, the root node is expanded first, then all the nodes generated by the root node are expanded next, and their successors, and so on. In general, all the nodes at depth *d* in the search tree are expanded before the nodes at depth *d+1*. Algorithmically: BFS(G,s) {

```
initialize vertices;
Q = {s};
while (Q not empty) {
    u = Dequeue(Q);
    for each v adjacent to u do {
        if (color[v] == WHITE) {
            color[v] = GRAY;
            d[v] = d[u]+1; // compute d[]
            p[v] = u; // build BFS tree
            Enqueue(Q,v);
        }
    }
    color[u] = BLACK;
}
```

BFS runs in O(V+E)

<u>Note</u>: BFS can compute d[v] = shortest-path distance from s to v, in terms of minimum number of edges from s to v (un-weighted graph). Its breadth-first tree can be used to represent the shortest-path.

Uniform-cost search expands the least-cost leaf node first. It is complete, and unlike breadth-first search is optimal even when operators have differing costs. Its space and time complexity are the same as for BFS. BFS finds the shallowest goal state, but this may not always be the least-cost solution for a general path cost function. UCS modifies BFS by always expanding the lowest-cost node on the fringe.

Depth-first search expands the deepest node in the search tree first. It is neither complete nor optimal, and has time complexity of O(b^m) and space complexity of O(bm), where m is the maximum depth. In search trees of large or infinite depth, the time complexity makes this impractical.
DFS always expands one of the nodes at the deepest level of the tree. Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower levels.

```
Algorithmically: DFS(G) {   for each vertex u in V     color[u] =
                WHITE;   time = 0; // global variable   for each vertex
                u in V     if (color [u] == WHITE)
                    DFS_Visit(u);
                }

                DFS_Visit(u) {   color[u] = GRAY;   time = time + 1; // global
                variable   d[u] = time; // compute discovery time d[]   for each v
                adjacent to u     if (color[v] == WHITE) {      p[v] = u; // build DFS-
                tree
                    DFS_Visit(u);
                   }
                  color[u] = BLACK;   time = time + 1; // global variable   f[u] =
                time; // compute finishing time f[]
                }
```

DFS runs in O(V+E)

DFS can be used to classify edges of G:

1. Tree edges: edges in the depth-first forest
2. Back edges: edges (u,v) connecting a vertex u to an ancestor v in a depth-first tree
3. Forward edges: non-tree edges (u,v) connecting a vertex u to a descendant v in a depth-first tree
   b. Cross edges: all other edges

An undirected graph is acyclic iff a DFS yields no back edges.

Depth-limited search
places a limit on how deep a depth-first search can go. If the limit happens to be equal to the depth of shallowest goal state, then time and space complexity are minimized. DLS stops to go any further when the depth of search is longer than what we have defined.

Iterative deepening search
calls depth-limited search with increasing limits until a goal is found. It is complete and optimal, and has time complexity of O(b^d)

IDS is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then depth 1, then depth 2, and so on. In effect, IDS combines the benefits of DFS and BFS.

Bidirectional search can enormously reduce time complexity, but is not always applicable. Its memory requirements may be impractical. BDS simultaneously search both forward form the initial state and backward from the goal, and stop when the two searches meet in the middle, however search like this is not always possible.

Graphs and their Applications – II

Introduction

In this unit the application of the graphs were discussed mainly on search algorithms. DFS, BFS and DF-ID were discussed with the analysis of examples like movement of nights in a nxn chess board. Different practical examples related were discussed. Informed search methods and application of graphs and the cost analysis algorithms are introduced.

Depth First Search (DFS) Algorithm

Form a one-element queue consisting of the root node. Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node. If the first element is the goal node, do nothing. If the first element is not the goal node, remove the first element from the queue and add the first element's children, if any, to the **front** of the queue. If the goal node has been found, announce success, otherwise announce failure.

Note: This implementation differs with BFS in insertion of first element's children, DFS from **FRONT** while BFS from **BACK**. The worst case for DFS is the best case for BFS and vice versa. However, avoid using DFS when the search trees are very large or with infinite maximum depths.

Sample Problem: n Queens [Traditional]

Place n queens on an n x n chess board so that no queen is attacked by another queen.

Depth First Search (DFS) Implementation

The most obvious solution to code is to add queens recursively to the board one by one, trying all possible queen placements. It is easy to exploit the fact that there must be exactly one queen in each column: at each step in the recursion, just choose where in the current column to put the queen.

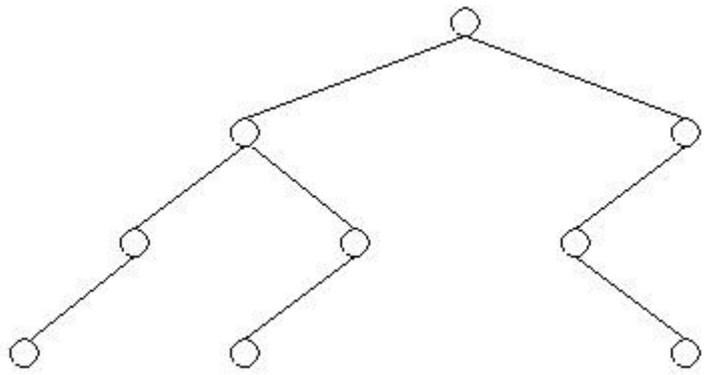
1. search(col)
2. if filled all columns
3. print solution and exit
4. for each row
5. if board(row, col) is not attacked
6. place queen at (row, col)
7. search(col+1)

8. remove queen at (row, col)

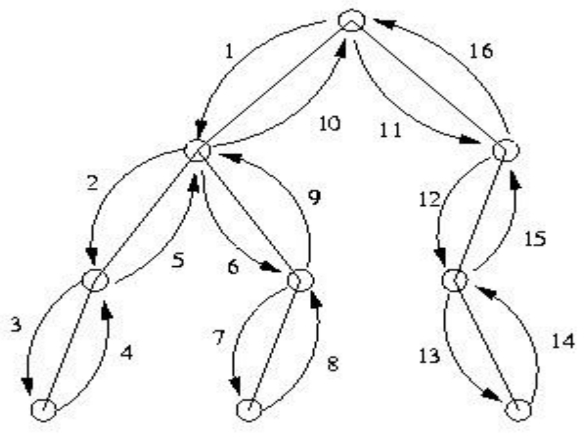
Calling search(0) begins the search. This runs quickly, since there are relatively few choices at each step: once a few queens are on the board, the number of non-attacked squares goes down dramatically.

This is an example of depth first search, because the algorithm iterates down to the bottom of the search tree as quickly as possible: once k queens are placed on the board, boards with even more queens are examined before examining other possible boards with only k queens. This is okay but sometimes it is desirable to find the simplest solutions before trying more complex ones.

Depth first search checks each node in a search tree for some property. The search tree might look like this:



The algorithm searches the tree by going down as far as possible and then backtracking when necessary, making a sort of outline of the tree as the nodes are visited. Pictorially, the tree is traversed in the following manner:



Complexity
Suppose there are d decisions that must be made. (In this case d=n, the number of columns we must fill.) Suppose further that there are C choices for each decision. (In this case c=n also, since any of the rows could potentially be chosen.) Then the entire search will take time proportional to c^d, i.e., an exponential amount of time. This scheme requires little space, though: since it only keeps track of as many decisions as there are to make, it requires only O(d) space.

Breadth First Search (BFS)
Form a one-element queue consisting of the root node. Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node. If the first element is the goal node, do nothing (or you may stop now, depends on the situation). If the first element is not the goal node, remove the first element from the queue and add the first element's children, if any, to the **back** of the queue. If the goal node has been found, announce success, otherwise announce failure.
The side effect of BFS:

1. Memory requirements are a bigger problem for BFS than the execution time
2. Time is still a major factor, especially when the goal node is at the deepest level.

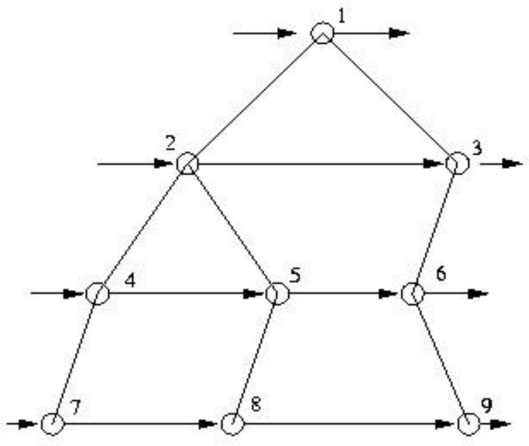
Sample Problem: Knight Cover [Traditional]
Place as few knights as possible on an n x n chess board so that every square is attacked. A knight is not considered to attack the square on which it sits.

Breadth First Search (BFS) Implementation
In this case, it is desirable to try all the solutions with only k knights before moving on to those with k+1 knights. This is called breadth first search. The usual way to implement breadth first search is to use a queue of states:

1. process(state)
2. for each possible next state from this one
3. enqueue next state
4. search()
5. enqueue initial state
6. while !empty(queue)
7. state = get state from queue

```
8.  process(state)
```

This is called breadth first search because it searches an entire row (the breadth) of the search tree before moving on to the next row. For the search tree used previously, breadth first search visits the nodes in this order:



It first visits the top node, then all the nodes at level 1, then all at level 2, and so on.

Complexity

Whereas depth first search required space proportional to the number of decisions (there were n columns to fill in the n queens problem, so it took O(n) space), breadth first search requires space exponential in the number of choices. If there are c choices at each decision and k decisions have been made, then there are c^k possible boards that will be in the queue for the next round. This difference is quite significant given the space restrictions of some programming environments.

Depth First with Iterative Deepening (DF-ID)

An alternative to breadth first search is iterative deepening. Instead of a single breadth first search, run D depth first searches in succession, each search allowed to go one row deeper than the previous one. That is, the first search is allowed only to explore to row 1, the second to row 2, and so on. This “simulates" a breadth first search at a cost in time but a savings in space.

```
1.  truncated_dfsearch (hnextpos, depth)
2.  if board is covered
3.  print solution and exit
4.  if depth == 0
```

```
5.  return
6.  for i from nextpos to n*n
7.  put knight at I
8.  search(i+1, depth-1)
9.  remove knight at i
10. dfid_search
11. for depth = 0 to max_depth
12. truncated_dfsearch(0, depth)
```

Complexity

The space complexity of iterative deepening is just the space complexity of depth first search: O(n). The time complexity, on the other hand, is more complex. Each truncated depth first search stopped at depth k takes ck time. Then if d is the maximum number of decisions, depth first iterative deepening takes c^0 + c^1 + c^2 + ... + c^d time.

If c = 2, then this sum is c^(d+1) - 1, about twice the time that breadth first search would have taken. When c is more than two (i.e., when there are many choices for each decision), the sum is even less: iterative deepening cannot take more than twice the time that breadth first search would have taken, assuming there are always at least two choices for each decision.

Comparison of DFS, BFS & DFS+ID

Once you've identified a problem as a search problem, it's important to choose the right type of search. Here are some information which may be useful while taking the decision. In a Nutshell

Search	Time	Space	When to use
DFS	O(c^k)	O(k)	Must search tree anyway, know the level the answers are on, or you aren't looking for the shallowest number.
BFS	O(c^d)	O(c^d)	Know answers are very near top of tree, or want shallowest answer.
DFS+ID	O(c^d)	O(d)	Want to do BFS, don't have enough space, and can spare the time.

d is the depth of the answer k is the depth searched d

<= k

Remember the ordering properties of each search. If the program needs to produce a list sorted shortest solution first (in terms of distance from the root node), use breadth first search or iterative deepening. For other orders, depth first search is the right strategy.

If there isn't enough time to search the entire tree, use the algorithm that is more likely to find the answer. If the answer is expected to be in one of the rows of nodes closest to the root, use breadth first search or iterative deepening. Conversely, if the answer is expected to be in the leaves, use the simpler depth first search. Be sure to keep space constraints in mind. If memory is insufficient to maintain the queue for breadth first search but time is available, use iterative deepening.